

Kapitel 4

Klassen und Objekte

In diesem Kapitel:

Einführung	48
Was ist eine Klasse?	52
Klassen anwenden	57
Klassencode	59
Zugriffsmodifizierer für Klassen, Methoden und Eigenschaften	67
Vererbung, Polymorphie, Abstrakte Klassen und Schnittstellen	70
Statische Komponenten	71
Delegaten und Lambda-Ausdrücke	73

Klassen sind der Hauptstützpfiler der objektorientierten Programmierung. Spricht man von Klassen und Klassenkonzepten, dann sind Technologien wie Schnittstellen, Klassenmodelle, das Ableiten bzw. Vererben von Klassen sowie Polymorphie mit dem gezielten Überschreiben von Klassenmethoden und –eigenschaften nicht weit. Doch darum soll es vordergründig in diesem Kapitel gar nicht gehen.

Klassen sind, mit Hinblick auf Auflistungen und letzten Endes auf das Datenabfragekonzept *LINQ to Objects*, in erster Linie mal Schablonen, die ihre Daten reglementieren und von der Außenwelt schützen – und auch nur unter diesem Gesichtspunkt sollen sie in diesem Kapitel erklärt werden.

Polymorphie im Rahmen von Klassenvererbung, insbesondere was das Überschreiben von Methoden und Eigenschaften anbelangt, das Erstellen von abstrakten Klassen für Klassenvorlagen und Schnittstellen sollen nicht Thema dieses Kapitels sein, da es für die weiteren Neuerungen, insbesondere LINQ nicht von so entscheidender Bedeutung ist (insgesamt aber natürlich schon – es würde nur einfach den Rahmen dieses Buches sprengen).

TIPP Wer unter Ihnen nach der Lektüre dieses Kapitels hungrig auf mehr geworden ist, der kann das komplette Buch *Visual Basic 2005 – das Entwicklerbuch* direkt von der ActiveDevelop-Homepage herunterladen. Sie erreichen unseren Internetauftritt unter <http://www.activedevelop.de>.

Einführung

»Eine Klasse schafft die Strukturen für das Speichern von Daten und beinhaltet gleichzeitig Programmcode, der diese Daten reglementiert.« Diese Erklärung finden Sie nicht nur oftmals als *die* Erklärung für das Konzept von Klassen, sie ist auch kurz, knackig, absolut zutreffend und so abstrakt, dass jemand, der sich zum ersten Mal mit dieser Materie beschäftigt, damit überhaupt nichts anfangen kann.

Deswegen wollen wir im Folgenden das Pferd bewusst von der anderen Seite aufzäumen und ein Beispiel bemühen, das zeigt, wie ein Programm Daten speichert und organisiert, das *ohne* Klassen zurecht kommen muss.

Miniadesso – die prozedurale Variante

Also lassen Sie uns mal einen Blick auf die »So-besser-Nicht«-Variante werfen – hochtrabend *Miniadesso-Prozedu* genannt.

HINWEIS Bei den Beispielprogrammen der folgenden Abschnitte handelt es sich nicht um Windows-, sondern um so genannte Konsolenanwendungen – um Anwendungen also, die sich ausschließlich unter der Windows-Eingabeaufforderung verwenden lassen.

Konsolenanwendung in VB.NET

Im Gegensatz zu Visual Basic 6.0 können Sie in Visual Basic .NET auch ohne größeren Aufwand Konsolenanwendungen entwerfen. Das sind Programme, die über keine grafische Oberfläche verfügen, sondern nur unter der Windows-Eingabeaufforderung laufen und mit dem Anwender über reine Textein- und -ausgabe kommunizieren. Konsolenanwendungen sind für Programme sehr gut geeignet, die bei der reinen Stapelverarbeitung eingesetzt werden sollen und wenig oder überhaupt keine Kommunikation mit dem Anwender erfordern. Aber gerade auch beim Debuggen – zum Beispiel, um ohne großen Aufwand neue Typen (Klassen, Strukturen) zu testen – leisten sie sehr gute Dienste. ▶

Möchten Sie selber eine neue Kommandozeilenanwendung erstellen, wählen Sie in der Visual Studio-IDE aus dem Menü *Datei* den Menüpunkt *Neu/Projekt* und im Dialog, den Visual Studio anschließend zeigt, aus dem Zweig *Visual Basic* und dem Bereich *Windows* die Vorlage *Konsolenanwendung*.

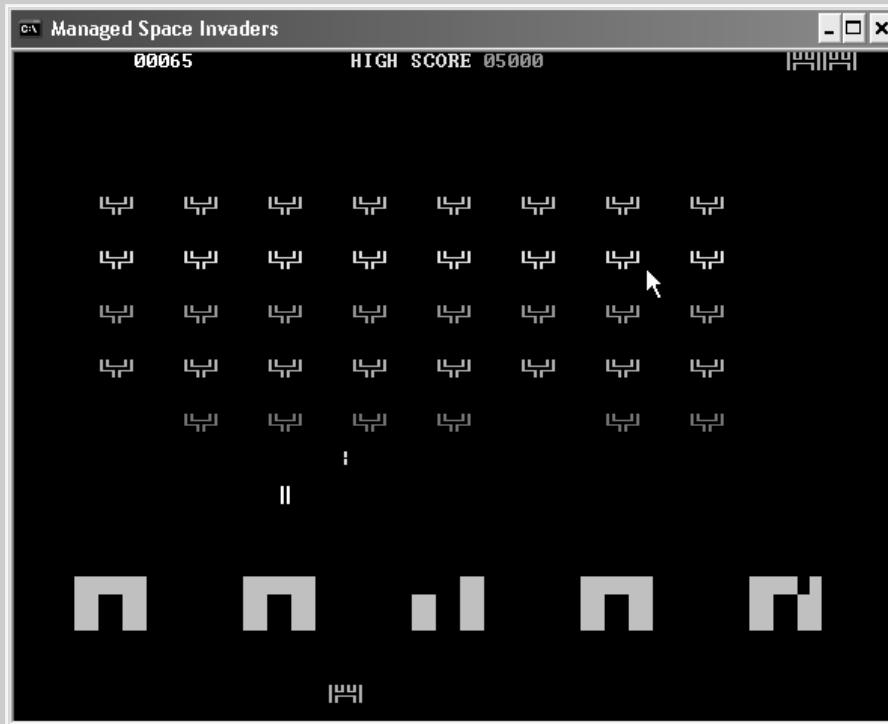


Abbildung 4.1 Kid George vom BCL-Team demonstrierte zum Launch von Visual Studio 2005 die Möglichkeiten von Konsolenanwendungen eindrucksvoll mit seiner Implementierung von Space Invaders!

Seit Visual Studio 2005, bzw. dem .NET Framework 2.0, gibt es übrigens einige Erweiterungen in der Kommandozeilenunterstützung aus .NET heraus. So haben Sie direkt über die *Console*-Klasse Einfluss auf die Farbgebung, auf die direkte Positionierung der jeweils nächsten Ausgabe und Sie können sogar ganze Bildschirmbereiche mit einfachen Befehlen verschieben, und damit Scrolling in jede Richtung oder sogar einfache bewegte Bildschirmgrafiken realisieren.

BEGLEITDATEIEN

Unter `.\Samples\Chapter04\MiniAdressoProzedu` finden Sie das Beispielprojekt dieses Abschnittes.

Wenn Sie das Programm starten, sehen Sie ein Konsolenfenster, das die Programmausgabe beinhaltet, wie Sie sie in etwa auch in Abbildung 4.2 betrachten können.

TIPP

Bei den vergleichsweise großen Auflösungen der 22 und 24 Zoll-Monitore, die heutzutage schon für kleines Geld zu haben sind, stellen Sie den Font der Konsolenfenster am besten auf die größte Schriftart (10 × 18), die Ihnen das Systemmenü über den Menüpunkt *Standardwerte* (Windows Vista) bzw. *Eigenschaften* (Windows XP) und im Dialog, der jetzt erscheint, über die Registerkarte *Schriftart* anbietet.

```

file:///C:/SharedProjects/Writing/VB Crashkurs/Samples/Chapter03/MiniAdressoProzedu/bin/Debug/MiniAdressoProzedu.EXE
Zufallsadressen werden generiert ... fertig!
000: Englisch, Christian, 51096, Lippetal
001: Weichel, Momo, 53649, Stirpe
002: Braun, Klaus, 18732, Bielefeld
003: Müller, Barbara, 00353, Soest
004: Vielstedde, Melanie, 99805, Soest
005: Ademmer, Barbara, 59598, Berlin
006: Tiemann, Guido, 38566, Dortmund
007: Heckhuis, Rainer, 20410, Bad Waldliesborn
008: Tinoco, Anne, 18989, Soest
009: Vielstedde, Guido, 75442, Lippetal
010: Weichel, Uta, 61062, Lippstadt

Adressen werden sortiert ... fertig!

000: Ademmer, Barbara, 51096, Berlin
001: Ademmer, Klaus, 51096, Lippetal
002: Albrecht, Theo, 51096, Berlin
003: Albrecht, Katrin, 51096, Berlin
004: Albrecht, Klaus, 51096, Hildesheim
005: Albrecht, Theo, 51096, Lippstadt
006: Braun, Rainer, 51096, Hildesheim
007: Braun, Franz, 53649, Lippstadt
008: Braun, Michaela, 51096, Bad Waldliesborn
009: Braun, Klaus, 53649, Stirpe
010: Englisch, José, 53649, Stirpe

Taste zum Beenden drücken...

```

Abbildung 4.2 Die erste Version des Programms macht scheinbar Dienst nach Vorschrift – doch durch das prozedurale Konzept hat sich leider ein schwerer Fehler eingeschlichen, der nicht gleich ersichtlich ist.

OK – hier haben wir also die erste Version unseres kleinen Beispielprogramms, das nichts weiter macht, als sich ein paar Kontaktadressen auszudenken, deren Einzeldaten in dafür vorgesehene Arrays abzulegen, die Kontakte nach Nachnamen zu sortieren und sie anschließend auszugeben. Schauen wir uns an, wie das Programm generell mit seinen Daten umgeht:

```

Module Hauptmodul

    Dim Nachname(0 To 100) As String
    Dim Vorname(0 To 100) As String
    Dim PLZ(0 To 100) As String
    Dim Ort(0 To 100) As String

    Sub Main()

```

Es benötigt für jede Kontaktadresse jeweils vier Einzelemente, die es in insgesamt vier Arrays ablegt, und genau das macht es so schwierig für den Entwickler, die Daten einer einzigen Kontaktadresse auch wirklich zusammen zu halten. Ohne eine *zusammenhängende* Entität zu schaffen, die die Daten eines Kontakts auch programmtechnisch mehr oder weniger automatisch bzw. durch ihr Konzept bedingt zusammenhält, sind Fehler buchstäblich vorprogrammiert – und genau ein solcher Fehler hat sich dann auch in dieser Programmversion eingeschlichen (OK, natürlich habe ich ihn zu Demozwecken einschleichen lassen, aber dieses prozedurale Programmierbeispiel ist typisch für eine interne Datenverwaltung, genau so typisch eben wie die Fehler, die dabei auftreten können).

```

Sub AdressenSortieren()

  'Beispiel für lokalen Typrückschluss
  Dim anzahlElemente = 101
  Dim delt = 1

  Dim aeussererZaehler As Integer
  Dim innererZaehler As Integer

```

Bei einem Blick auf die Sortierroutine fällt zunächst etwas Besonderes auf, was es neu in Visual Basic 2008 gibt und unter dem Namen *lokaler Typrückschluss* bezeichnet wird. Dabei werden die Typen von Variablen nur durch reine Zuweisung von Konstanten festgelegt. Kapitel 3 erklärt, wie der lokale Typrückschluss genau funktioniert.

Doch zurück zum eigentlichen Ausgangsproblem: Die Sortieroutine des Programms ist in diesem Beispiel der Übeltäter: Gerade sie müsste durch extrem sorgfältige Ausführung bei der Programmierung dafür sorgen, dass ein Adressdatensatz logisch zusammengehörig bleibt, damit die Integrität des gesamten Datenaufkommens nicht verletzt wird. Schauen wir uns diese Sortieroutine jedoch genauer an ...

```

  Dim tempVorname, tempNachname, tempPLZ, tempOrt As String

  'Größten Wert der Distanzfolge ermitteln
  Do
    delta = 3 * delta + 1
  Loop Until delta > anzahlElemente

  Do
    'Späteres Abbruchkriterium - entsprechend kleiner werden lassen
    delta \= 3

    'Shellsort's Kernalgorithmus
    For aeussererZaehler = delta To anzahlElemente - 1
      tempVorname = Vorname(aeussererZaehler)
      tempNachname = Nachname(aeussererZaehler)
      tempPLZ = PLZ(aeussererZaehler)
      tempOrt = Ort(aeussererZaehler)

      innererZaehler = aeussererZaehler
      Do
        If tempNachname >= Nachname(innererZaehler - delta) Then Exit Do
        Vorname(innererZaehler) = Vorname(innererZaehler - delta)
        Nachname(innererZaehler) = Nachname(innererZaehler - delta)
        PLZ(innererZaehler) = PLZ(innererZaehler - delta)

        innererZaehler = innererZaehler - delta
        If (innererZaehler <= delta) Then Exit Do
      Loop
      Vorname(innererZaehler) = tempVorname
      Nachname(innererZaehler) = tempNachname
      Ort(innererZaehler) = tempOrt
    Next
  Loop Until delta = 0
End Sub

```

... so stellen wir bei näherem Hinsehen fest, dass sich, was die Datenintegrität anbelangt, ein dicker Fehler in den Algorithmus eingeschlichen hat: Beim Dreieckstausch eines Kontaktelements nämlich, werden nicht wirklich alle Felder getauscht. Beim »Hintauschen« bleibt der Ort, beim »Zurücktauschen« die Postleitzahl auf der Strecke; für ein kommerzielles Programm wäre das natürlich eine Katastrophe, zumal selbst bei einer Liste von nur 100 Kontaktadressen dieser Fehler nicht sofort ins Auge sticht.

Was ist eine Klasse?

Versetzen Sie sich in Ihre Kindheit zurück. Und zwar so weit, dass Sie sich vorstellen können, im Sandkasten zu sitzen und mit Förmchen und nassem Sand zu spielen. Sie werden es nicht glauben, aber genau zu diesem Zeitpunkt haben Sie bereits das Klassenkonzept angewendet. Ein Förmchen ist im Grunde genommen nämlich nichts anderes als eine Klasse. Das Förmchen gibt vor, wie Objekte aussehen sollen, die aus ihm entstehen werden, aber selbst ist es noch kein Objekt, sondern nur eine Vorlage. Wenn Sie aus einer Klasse (einem Förmchen) einen Sandkuchen (ein Objekt) machen wollen, dann müssen Sie ein Objekt dieser Klasse instanziiieren. Um bei der Analogie zu bleiben: Sie instanziiieren einen Sandkuchen aus einem Förmchen, indem Sie nassen Sand in die Form hineingeben, das ganze Ding umdrehen und die Form abziehen.

Klassen im .NET Framework sind natürlich etwas abstrakter, aber Sie machen sich daran auch nicht die Hände schmutzig. Sie bestehen erst einmal aus einer Reihe von sogenannten Member-Variablen – auch Felder oder Feldvariablen genannt – und der Klassenrumpf-Definition, die diese Variablen einschließt und der Klasse ihren Namen gibt.

Damit existiert dann bereits die einfachste Version einer Klasse, die, wie wir gleich sehen werden, ihre Daten logisch kapselt. Immerhin erlaubt sie uns schon, das Problem unseres ersten Beispielprogramms im Ansatz zu ersticken – die zweite Version dieses Demoprogramms stellt das unter Beweis. Hier gibt es nun eine Klasse namens *Kontakt*, die aus vier Feldern besteht – die vier Datenfelder, die eine einzige Adresse ausmachen. In der Folge benötigen wir nun nicht mehr vier, eigentlich völlig zusammenhanglose Arrays, sondern nur noch ein einziges Array¹, das aus Elementen besteht, die jeweils eine komplette Adresse speichern.

Um das zu tun, fügen wir dem bestehenden Programm einfach eine neue Codedatei hinzu – eine Klassen-datei – namens *Kontakt.vb*. In diese Klassen-Codedatei schreiben wir dann folgende Zeilen:

```
Public Class Kontakt
    Public Nachname As String
    Public Vorname As String
    Public PLZ As String
    Public Ort As String
End Class
```

Diese Klasse namens *Kontakt* stellt die einfachste Form einer Klasse dar. Sie enthält nur vier öffentlich zugängliche Felder, aber die Klasse fasst eben diese vier Felder logisch zu einem Datensatz zusammen.

¹ Mehr zu Arrays und Auflistungen erfahren Sie im nächsten Kapitel.

Klassen mit New instanziiieren

Um eine so genannte Instanz dieser Klasse zu schaffen, bedient man sich des Schlüsselwortes `New` – intern wird damit der entsprechende Speicherplatz geschaffen, den man benötigt, um die Daten der Member-Variablen der Klasse im Arbeitsspeicher² abzulegen. Eine Objektvariable, die man zuvor als Typ dieser Klasse definiert, dient dann dazu, auf die Elemente der Klasseninstanz zuzugreifen. Die Anwendung dieser einfachen Beispielklasse würde also wie folgt aussehen:

```
Dim adr As Kontakt  
adr = New Kontakt
```

Die erste Codezeile legt eine Objektvariable vom Typ `Kontakt` an und die zweite Codezeile weist ihr eine neue Instanz zu. Sie können diese beiden Zeilen auch in einer Zeile zusammenfassen:

```
Dim adr As New Kontakt
```

Anschließend verwenden Sie die Instanz dieser Klasse mithilfe der definierten Objektvariablen, um auf die einzelnen Felder zuzugreifen:

```
adr.Nachname = "Dröge"  
adr.Vorname = "Ute"  
adr.PLZ = "59555"  
adr.Ort = "Lippstadt"
```

Um nochmals auf unsere Analogie mit dem Sandkuchen zurückzukommen: Die Objektinstanz entspricht hier einem Sandkuchen, der aus einem Förmchen hervorgegangen ist. Die Klasse hingegen ist das Förmchen. Die Objektvariable benennt wiederum den Sandkuchen, der aus dem Förmchen hervorgegangen ist, und natürlich können Sie, wie aus dem Förmchen, beliebig viele Sandkucheninstanzen erstellen. Beliebige viele? Nicht ganz: Natürlich nur so viele, bis der »verwaltete Sandberg« zuneige geht, aus dem Sie die Sandkuchen machen, der in der Analogie dem Managed Heap entspricht, aus dem Sie den Speicherplatz für Ihre Objektinstanzen beziehen.

Ein gern gemachter Fehler am Anfang ist es übrigens, die Klasse nicht zu instanziiieren, sondern direkt verwenden zu wollen:

```
'So geht's natürlich nicht!  
Kontakt.Nachname = "Dröge"  
Kontakt.Vorname = "Ute"  
Kontakt.PLZ = "59555"  
Kontakt.Ort = "Lippstadt"
```

Warum? – Denken Sie mal drüber nach. In Analogie zu einer primitiven Variablen würden Sie praktisch statt

² Genau genommen in einem bestimmten Teil des Arbeitsspeichers, der durch das .NET Framework verwaltet, d.h. ständig aufgeräumt wird. Sein Name: *Managed Heap*.

```
Dim i as Integer
i = 5
```

einfach

```
Integer = 5
```

schreiben, quasi den Bezeichner für den Datentyp als Variablenamen verwenden. Das geht natürlich nicht!

Öffentliche Felder oder Eigenschaften beim Instanzieren initialisieren

Visual Basic erlaubt neu in der 2008er Version auch eine kürzere Variante beim Vorbelegen der Felder bzw. Eigenschaften mit Werten: Sie können öffentliche Felder oder Eigenschaften direkt beim Instanzieren mithilfe des `With`-Schlüsselwortes definieren, wie im folgenden Beispiel zu sehen:

```
Dim adr As New Kontakt With {.Nachname = "Dröge", _
                             .Vorname = "Ute", .PLZ = "59555", .Ort = "Lippstadt"}
```

New oder nicht New – wieso es sich bei Objekten um Verweistypen handelt

Zum besseren Verständnis für alle späteren Kapitel ist es überaus sinnvoll, ein paar Worte zur Speicherung von Objekten, also Instanzen von Klassen zu verlieren. Objektvariablen und Objekte sind nämlich nicht *so* miteinander verbunden, wie man es sich zunächst vielleicht vorstellt. Im Gegenteil: Der Verbund einer Objektvariablen mit den eigentlichen Daten des Objektes hält bestenfalls so gut, wie eine amerikanische Prominentenehe: Was auf den ersten Blick so innig und für immer geschaffen zu sein scheint, ist in der nächsten Sekunde auch schon wieder sauber getrennter Schnee von gestern.

Die Wahrheit ist nämlich: Eine Objektvariable speichert im Grunde genommen nur *einen Zeiger* auf die eigentlichen Daten im Managed Heap, in den die Daten der Objekte gelangen, die mithilfe von `New` aus Klassen instanziiert werden.

Und wie wir (die Älteren unter uns jedenfalls) aus unseren Anfängertagen mit 64ern, Atari STs und Maschinensprache noch alle wissen, untergliedert sich der Arbeitsspeicher eines Computers in bestimmte Speicherstellen, die alle bestimmte »Hausnummern« (die Speicheradressen) besitzen.

Wenn Sie nun ein Objekt aus einer Klasse erstellen – zum Beispiel indem Sie einen `Kontakt`-Datentyp³ mit `New` in die Objektvariable `objVarKontakt` instanzieren – dann legt das .NET Framework die Daten für diese Objektinstanz beispielsweise an Speicheradresse 460.386 auf dem Managed Heap ab, und die Objektvariable wird intern sozusagen zu einer Integer-Variablen (oder auf 64-Bit-Systemen zu einer Long-Variablen), die diese Adresse trägt. Bildlich sieht das folgendermaßen aus:

³ Für diese Erklärung wurde übrigens auch der Datentypname `Kontakt` dem Namen `Adresse` vorgezogen, um nicht zu viel Konfusion mit dem Wort *Speicheradresse* entstehen zu lassen.

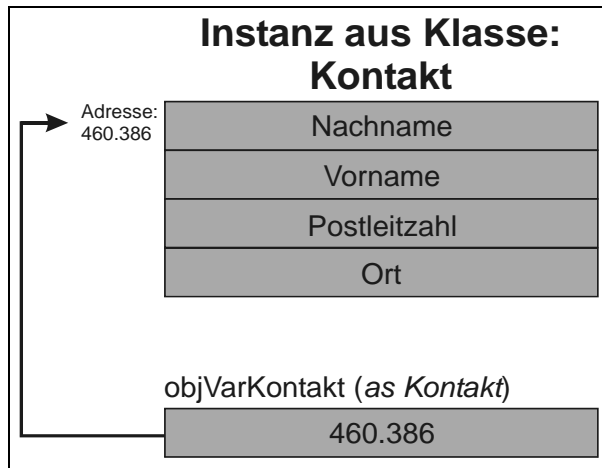


Abbildung 4.3 Objektvariablen speichern im Grunde genommen nur die Speicheradressen auf die eigentlichen Daten, die das .NET Framework im Managed Heap ablegt

Diese Tatsache hat aber entscheidende Folgen, wie das folgende Beispiel gleich zeigen wird.

BEGLEITDATEIEN

Sie finden dieses Beispielprojekt unter `.\Samples\Chapter04\KlassenObjekteSpeicher`.

Module Module1

Sub Main()

```
'Instanzieren mit New und dadurch
'Speicher für das Kontakt-Objekt
'auf dem Managed Heap anlegen.
'Instanz gleichzeitig mit Daten initialisieren.
Dim objVarKontakt As New Kontakt With {.Nachname = "Pfeiffer", _
    .Vorname = "Ute", .PLZ = "59555", .Ort = "Lippstadt"}
```

```
'Nur Objektvariable anlegen,
'es wird aber kein Speicher reserviert!
Dim objVarKontakt2 As Kontakt
```

```
'objVarKontakt2 "zeigt" ab jetzt auf
'dieselbe Instanz wie objVarKontakt
objVarKontakt2 = objVarKontakt
```

```
'Und das kann man auch beweisen:
'Das Ändern der Instanz geschieht...
objVarKontakt2.Nachname = "Dröge"
```

```
'durch beide Objektvariablen, die natürlich
'auch dasselbe widerspiegeln.
Console.WriteLine(objVarKontakt.Nachname)
```

```
Console.WriteLine()
Console.WriteLine("Zum Beenden Taste drücken")
Console.ReadKey()
```

End Sub

```

End Module

Public Class Kontakt
    Public Vorname As String
    Public Nachname As String
    Public PLZ As String
    Public Ort As String
End Class

```

Wenn Sie dieses Beispiel laufen lassen, erhalten Sie die Ausgabe

```

Dröge
Zum Beenden Taste drücken

```

Beachten Sie, dass es in diesem Beispiel zwar nur eine einzige Dateninstanz der Klasse Kontakt gibt, die jedoch durch zwei Objektvariablen angesprochen und damit auch wiedergespiegelt wird: Beim Instanzieren wird die Adresse des Speichers, an dem die Daten der Instanz abgelegt werden, in der Objektvariablen objVarKontakt gespeichert. Diese Adresse wird in die Variable objVarKontakt2 übertragen, und wichtig, hierbei wird nicht etwa eine Kopie der gesamten Instanz im Managed Heap angelegt!

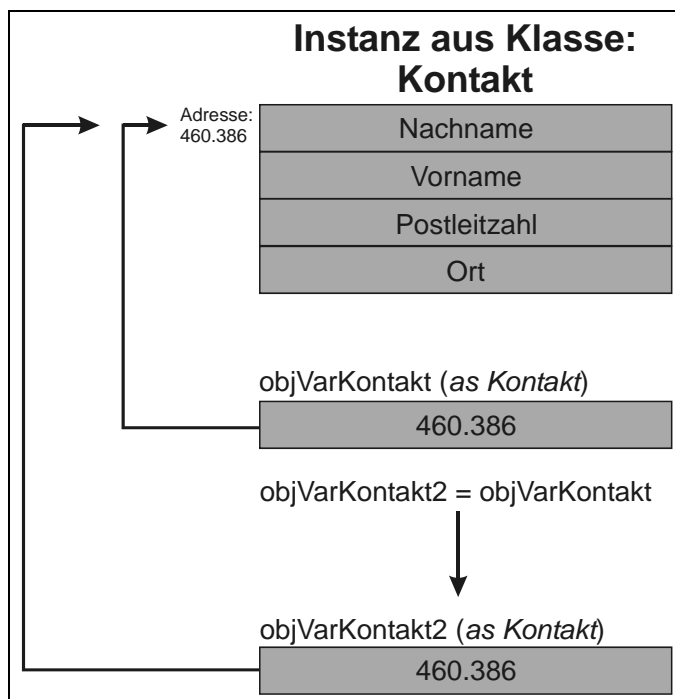


Abbildung 4.4 Das Kopieren einer Objektvariablen in eine andere Objektvariable kopiert nur den Zeiger auf die Instanz – die dann durch beide Variablen manipulierbar und abrufbar wird

Eine Objektvariable »zeigt« also quasi auf die Daten, die sie verwaltet. In anderen Programmiersprachen wie C++ gibt es zu diesem Zweck besondere Variablen, die auch als *Zeiger* bezeichnet werden. In Visual Basic wird eine Objektvariable, die die Instanz einer Klasse verwaltet, automatisch zu dem, was man in anderen Programmiersprachen als Zeiger bezeichnet. In Form einer Grafik sieht das Ganze dann aus wie in Abbildung 4.4.

Dieses Verhältnis zwischen Objektvariable und eigentlichem Objekt (eigentlicher Instanz) sollten Sie sich gut einprägen, da sie einerseits Quelle schwer zu findender Fehler ist – schließlich kann es auch versehentlich passieren, dass, wenn Sie nicht aufpassen, eine Objektvariable die Instanz eines Objektes verändert, die eigentlich und ausschließlich durch eine ganz andere Objektvariable angesprochen werden sollte. Andererseits kann Ihnen dieses Verhältnis auch zum Vorteil gereichen, nämlich wenn es darum geht, nicht Objekte zu kopieren, sondern nur die Zeiger auf diese – beispielsweise wenn es beim Sortieren großer Objektmengen auf Geschwindigkeit ankommt und deswegen keine ganzen Speicherblöcke mit den eigentlichen Daten sondern nur die Zeiger auf die Objektinstanzen kopiert werden sollen.

Nothing

Mit dem Wissen des vorherigen Abschnittes können Sie sich auch erklären, wieso eine Objektvariable den Wert `Nothing` (`null` in C# - nur der Vollständigkeit halber erwähnt) aufweisen kann. Wenn Sie mit `New` eine Instanz einer Klasse erstellen, weisen Sie diese Instanz (genauer: die Adresse dieser Instanz) der Objektvariablen zu. Erst dann können Sie mithilfe der Objektvariablen auf die öffentlichen Felder, Eigenschaften und Methoden der Klasseninstanz (des Objektes) zugreifen.

Wenn Sie versuchen mit einer nicht initialisierten Objektvariablen auf Klassen-Member zuzugreifen, wird zur Laufzeit eine entsprechende Ausnahme ausgelöst. In vielen Fällen kann der Hintergrundcompiler von Visual Basic diesen Zustand voraussehen und zeigt eine entsprechende Warnmeldung an (siehe Abbildung 4.5).

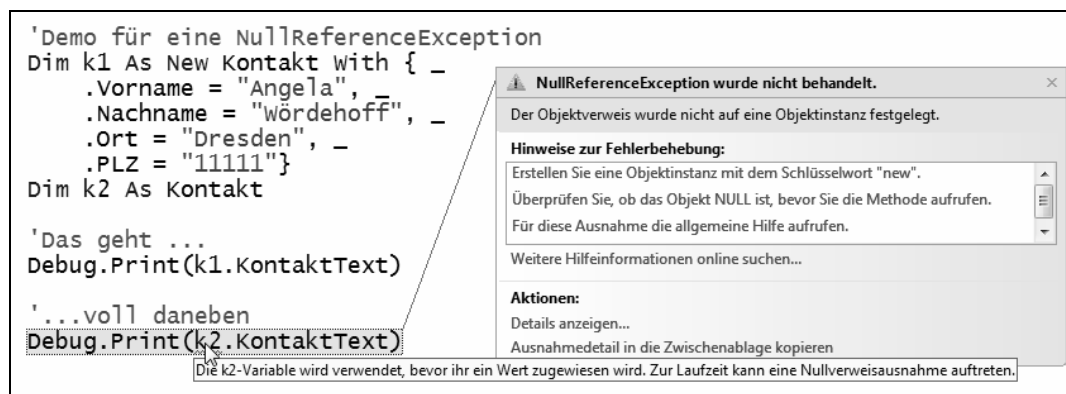


Abbildung 4.5 Wenn Sie mit einer Objektvariablen, der keine Klasseninstanz zugeordnet ist, auf Klassenelemente (Felder, Eigenschaften, Methoden) zugreifen, wird eine `NullReferenceException` ausgelöst

Klassen anwenden

Natürlich handelt es sich bei Klassen zwar um ganz neue Datentypen, die Sie aber genauso handhaben können, wie alle anderen Datentypen des .NET Frameworks. Und damit können einzelne Instanzen einer Klasse genauso in Arrays abgelegt werden, wie das bei ganz normalen primitiven Datentypen der Fall ist – wir sind damit beim ersten richtigen Anwendungsbeispiel für Klassen, nämlich der zweiten Version unseres Adressenbeispiels.

Mit all den Infos aus den vorherigen Abschnitten können wir das anfängliche Beispielprogramm nun so umschreiben, dass es »nur« noch mit einem Array auskommt, in dem Elemente vom gerade neu geschaffenen Typ Kontakt vorhanden sind, und dazu sind nicht einmal viele Änderungen notwendig:

BEGLEITDATEIEN

Sie finden dieses Beispielprojekt unter `.\Samples\Chapter04\MiniAdressoClass`.

Die erste betrifft das Einrichten der Datenstruktur am Anfang des Programms:

```
Module Hauptmodul
```

```
    Dim Kontakte(0 To 100) As Kontakt
```

Im Gegensatz zur ursprünglichen Version kommen wir jetzt mit nur noch einem Array aus, das Elemente enthält, die die Felder eines kompletten Kontaktes kapseln. Viel aufgeräumter sieht nun auch der Sortieralgorithmus aus (Änderungen an ihm sind fett hervorgehoben).

```
Sub AdressenSortieren()

    Dim anzahlElemente As Integer = 101

    Dim aeussererZaehler As Integer
    Dim innererZaehler As Integer
    Dim delta As Integer

    Dim tempKontakt As Kontakt

    delta = 1

    'Größten Wert der Distanzfolge ermitteln
    Do
        delta = 3 * delta + 1
    Loop Until delta > anzahlElemente

    Do
        'Späteres Abbruchkriterium - wieder kleiner werden lassen
        delta = delta \ 3

        'Shellsort's Kernalgorithmus
        For aeussererZaehler = delta To anzahlElemente - 1
            tempKontakt = Kontakte(aeussererZaehler)

            innererZaehler = aeussererZaehler
            Do
                If tempKontakt.Nachname >= Kontakte(innererZaehler - delta).Nachname Then Exit Do
                Kontakte(innererZaehler) = Kontakte(innererZaehler - delta)

                innererZaehler = innererZaehler - delta
                If (innererZaehler <= delta) Then Exit Do
            Loop
            Kontakte(innererZaehler) = tempKontakt
        Next
    Loop Until delta = 0
End Sub
```

Da der neue Datentyp `Kontakt` die Daten einer kompletten Kontaktadresse kapselt, kann der Fehler aus der letzten Programmversion gar nicht mehr passieren. Der Dreieckstausch vollzieht sich jetzt zwangsläufig an einem vollständigen Kontakt. Somit ist die Gefahr ausgeschlossen, dass einzelne Felder auf der Strecke bleiben. Die fettgedruckten Zeilen im vorherigen Listing dokumentieren das »Aufräumen«.

Klassencode

Betrachten wir nochmals die anfängliche Definition von Klassen: »Eine Klasse schafft die Strukturen für das Speichern von Daten und beinhaltet gleichzeitig Programmcode, der diese Daten reglementiert.« Bislang haben wir erst die eine Komponente einer Klasse kennen gelernt: so genannte Member-Variablen, die die Möglichkeit schaffen, bestimmte Dinge in einer Klasseninstanz zu speichern.

Der zweite Teil, der den Zugriff auf die Daten reglementieren soll, fehlt bislang. Das Klassenkonzept sieht es vor, dass datenreglementierender Code in Form von zwei Prozedurtypen in einer Klasse vorhanden sein kann: In Methoden (Sub, Function) und in Eigenschaftenprozeduren (Property).

Eigenschaftenprozeduren (Property-Prozeduren)

Die bislang einzige Beispielklasse dieses Kapitels bestand aus nur vier Member-Variablen, auf die man, da sie mit dem Schlüsselwort `public` (also als öffentlich zugänglich) definiert wurden – in jeder Instanz zugreifen konnte. Das hat mit der Kapselung von Daten natürlich denkbar wenig zu tun; dieses Beispiel diente bislang lediglich dazu, verschiedene Daten zu einer logischen Dateneinheit zusammenzufassen – von wirklicher Kapselung der Daten, bei der das Rein und Raus der Daten durch irgendeine Klassenpolizei geregelt wurde, konnte man dabei allerdings noch nicht sprechen.

Jetzt stellen Sie sich vor, Sie möchten, dass die letzten drei Buchstaben des Ortes, falls er mehr als 30 Buchstaben hat, durch drei Auslassungspunkte »...« ersetzt wird.

In der prozeduralen Welt würden Sie dazu eine Funktion entwickeln, die überprüft, ob eine Zeichenfolge mehr als dreißig Zeichen aufweist. Wäre das der Fall, würden Sie sie jedes Zeichen nach dem 30. Zeichen abschneiden sowie die letzten drei Zeichen der Zeichenkette durch die drei Auslassungspunkte ersetzen lassen.

Sie würden dann nicht direkt auf die Feldvariable zugreifen, sondern, wie im folgenden Beispiel, jeden Zugriff auf das Feld *Ort* in einen Funktionsaufruf einbauen. Das könnte dann im Ergebnis folgendermaßen ausschauen:

Wir haben also zunächst unsere »AuslassungszeichenAnText«-Routine ...

```
Function EllipseString(ByVal text As String, ByVal MaxLength As Integer) As String
    Dim tmpText As String

    If text.Length > MaxLength Then
        tmpText = text.Substring(0, MaxLength - 3) + "... "
    Else
        tmpText = text
    End If
    Return tmpText
End Function
```

... und diese Routine rufen Sie immer dann auf, wenn Sie auf das kritisch zu behandelnde Feld *Ort* einer Instanz der Klasse *Kontakt* zugreifen:

```

Sub KlasseVerarbeiten()
.
.
.
    Dim tmpKontakt As New Kontakt
.
.
.
    tmpKontakt.Ort = EllipseString(Console.ReadLine, 30)
End Sub

```

Es gibt aber noch eine viel elegantere und vor allen Dingen *sichere* Möglichkeit, dafür zu sorgen, dass die Member-Variable `Ort` der `Kontakt`-Klasse niemals eine zu lange Zeichenkette zugewiesen bekommt: So genannte *Eigenschaften-Prozeduren*.

Wenn Sie schon längere Zeit mit Visual Basic (egal, ob mit .NET oder 6.0) programmieren, haben Sie Eigenschaften natürlich längst kennen gelernt. Mithilfe von Eigenschaften können Sie in der Regel bestimmte Zustände von Objekten abfragen *und* verändern, und von außen »fühlt« sich das dann so an, also würden Sie direkt ein öffentliches Feld einer Klasse manipulieren oder abfragen. Möchten Sie beispielsweise wissen, ob die Schaltfläche eines Formulars anwählbar ist, verwenden Sie die Eigenschaft in Abfrageform, etwa wie hier:

```
If Schaltfläche.Enabled Then TuWas
```

Oder Sie legen die Eigenschaft eines Objektes fest, etwa wie mit der folgenden Zeile:

```
Schaltfläche.Enabled = false ' Abschießen, kommt keiner mehr 'ran
```

Sie wissen nun aber aus Erfahrung, dass Sie beim Setzen einer Schaltfläche nicht nur den Zustand einer Variablen dieser Button-Instanz verändern, sondern dass das Setzen dieser Eigenschaft auch noch etwas Weiteres bewirkt – im Fallbeispiel nämlich, dass die Schaltfläche auch sichtbar auf dem Formular ausgegraut wird (oder eben wieder – beim Zuweisen von `True` – den visuellen Einschaltzustand bekommt). Der Code, der dafür sorgt, muss innerhalb der Klasse natürlich irgendwo platziert werden, und so lautet die viel interessantere Frage: Wie statten Sie Ihre eigenen Klassen mit Eigenschaften aus?

Visual Basic stellt Ihnen zu diesem Zweck, wie schon erwähnt, Eigenschaftenprozeduren zur Verfügung. Eine Eigenschaft wird in Visual Basic innerhalb einer Klasse folgendermaßen definiert:

```

Property EineEigenschaft() As Datentyp

    Get
        Return DatentypObjektvariable
    End Get

    Set(ByVal Value As Datentyp)
        DatentypObjektvariable = Value
    End Set

End Property

```

Wenn Sie diese Eigenschaft in einer Klasse implementieren, können Sie sie bei instanziierten Objekten dieser Klasse auf folgende Weise verwenden:

Zuweisen von Eigenschaften

Mit der Anweisung

```
Object.EineEigenschaft = Irgendetwas
```

weisen Sie der Eigenschaft `EineEigenschaft` des Objektes einen Wert zu. Sie können im *Set-Accessor*⁴ (`Set(ByVal Value As Datentyp)`) der Eigenschaftensprozedur mit `Value` auf das Objekt zugreifen, das sich in `Irgendetwas` befindet. Nur der *Set*-Teil der Eigenschaftensprozedur wird in diesem Fall ausgeführt.

Ermitteln von Eigenschaften

Umgekehrt können Sie mit der Anweisung

```
Irgendetwas = Object.EineEigenschaft
```

den Inhalt der Eigenschaft wieder auslesen. In diesem Fall wird nur der *Get-Accessor* der Eigenschaftensprozedur ausgeführt, die das Ergebnis mit `Return` zurückliefert und dieses der Objektvariablen `Irgendetwas` zuweist, die natürlich vom gleichen Typ, wie die Eigenschaft sein muss.

Mit diesem Wissen können wir unsere Klasse jetzt folgendermaßen umschreiben:

BEGLEITDATEIEN

Sie finden dieses Beispielprojekt unter `.\Samples\Chapter04\MiniAdressoClass V2`.

```
Public Class Kontakt
    Private myVorname As String
    Private myNachname As String
    Private myPLZ As String
    Private myOrt As String

    Public Property Vorname() As String
        Get
            Return myVorname
        End Get
        Set(ByVal value As String)
            myVorname = value
        End Set
    End Property

    Public Property Nachname() As String
        Get
            Return myNachname
        End Get
        Set(ByVal value As String)
            myNachname = value
    End Property
End Class
```

⁴ Etwa: »Zugreifer«.

```

        End Set
    End Property

    Public Property PLZ() As String
        Get
            Return myPLZ
        End Get
        Set(ByVal value As String)
            myPLZ = value
        End Set
    End Property

    Public Property Ort() As String
        Get
            Return myOrt
        End Get
        Set(ByVal value As String)
            myOrt = EllipseString(value, 30)
        End Set
    End Property

    Private Function EllipseString(ByVal text As String, ByVal MaxLength As Integer) As String

        Dim tmpText As String

        If text.Length > MaxLength Then
            tmpText = text.Substring(0, MaxLength - 3) + "... "
        Else
            tmpText = text
        End If
        Return tmpText
    End Function

End Class

```

Sie sehen an diesem einfachen Beispiel, wie der Schutz der eigentlichen Datenstruktur durch Eigenschaftenprozeduren vonstatten geht: Die Daten, die in einer Klasseninstanz gespeichert werden, weisen private Zugriffsmodifizierer auf (mehr zu Zugriffsmodifizierern finden Sie im Abschnitt »Zugriffsmodifizierer für Klassen, Methoden und Eigenschaften« ab Seite 67), und sie können damit nicht mehr direkt von außen, sondern nur noch durch den Klassencode selbst verändert werden. Die Member-Variablen heißen auch nicht mehr wie zuvor, sondern tragen den Zusatz bzw. das Präfix »my« als rein textlichen Hinweis darauf, dass es sich dabei um »meine« Variablen aus Klassen- bzw. Klasseninstanzsicht handelt. Dies ist eine durchaus übliche, aber nicht vorgeschriebene Konvention – Sie können die Benennung der Member-Variablen nach eigenem Belieben vornehmen; Sie sollten allerdings darauf achten, dass Sie die Member-Variablen anders als Ihre von außen zugreifbaren Eigenschaften nennen, denn das würde zu einem Kompilierungsfehler führen.

Die Eigenschaftenprozeduren schließlich heißen so, wie die Namen der ursprünglich öffentlichen Member-Variablen. Aus Entwicklersicht hat sich damit nichts an der Handhabung geändert – im Ergebnis allerdings schon, denn durch die Eigenschaftenroutinen wird nun der Zugriff auf die Member-Variablen exklusiv geregelt. In diesem kleinen Beispiel sehen Sie, dass der Ort durch dieses Verfahren niemals mehr als 30 Zeichen aufweisen wird. Bei mehr als 30 Zeichen wird durch das Einbinden der `EllipseString`-Methode die Anpas-

sung der Zeichenkette mit den Auslassungszeichen vorgenommen und auf den eigentlichen Datenspeicher des Orts – die Member-Variable `myOrt` – kann von außen niemand mehr zugreifen, da ihr Zugriff durch Verwendung des Zugriffsmodifizierers `private` für diese (wie auch für alle anderen Member-Variablen) geschützt ist. Genau das ist es, was eines der Hauptanliegen einer Klasse ist, nämlich *Daten zu kapseln und den Zugriff auf sie durch Klassencode zu reglementieren*.

Öffentliche Variablen oder Eigenschaften – eine Glaubensfrage?

Jetzt haben Sie schon so viel über Eigenschaften erfahren – vielleicht fragen Sie sich, wieso man sie anstelle von einfachen öffentlichen Member-Variablen einsetzen sollte.

Solange, wie Sie Eigenschaften in einer Klasse nur benötigen, um irgendwelche Werte zu speichern, aber beim Abfragen oder Setzen dieser Werte nichts Weiteres passieren muss, wären öffentliche Variablen eigentlich ausreichend.

Die Klasse

```
Class MitEinerEigenschaft
    Public DieEigenschaft As Integer
End Class
```

erfüllt zunächst nämlich den gleichen Zweck wie die folgende Klasse,

```
Class AuchMitEinerEigenschaft

    Private myDieEigenschaft As Integer

    Public Property DieEigenschaft() As Integer
        Get
            Return myDieEigenschaft
        End Get
        Set(ByVal Value As Integer)
            myDieEigenschaft = Value
        End Set
    End Property
End Class
```

die natürlich sehr viel mehr Schreibarbeit erfordert. Prinzipiell ist das richtig. Und dennoch ist die zweite Methode der ersten Methode vorzuziehen, aus folgenden Gründen, die Ihnen teilweise schon bekannt sind:

- Eigenschaften kapseln und reglementieren den Zugriff auf Daten, wie Sie im vorangegangenen Beispiel bereits gesehen haben. Auch wenn Sie eine Eigenschaft nur komplett an eine Member-Variable durchreichen – für mögliche Erweiterungen, was Reglementierungsalgorithmen anbelangt, sind Sie auf jeden Fall schon mal vorbereitet.
- Felder lassen sich grundsätzlich nicht für die Datenbindung verwenden. Vielfach werden nicht nur Quellen wie Datenbanken, sondern auch Objekte, wie beispielsweise Auflistungen (Collections) als Datenquellen verwendet. Öffentliche Felder (also öffentliche Member-Variablen) können dabei nicht als Datenquelle dienen.


- Ein weiteres wichtigeres Argument ist das Ersetzen von Eigenschaften durch die so genannte Polymorphie beim Vererben von Klassen, auf das wir im Rahmen dieses Crashkurses leider nicht eingehen können. Dennoch soviel: Eine einmal als öffentlich deklarierte Variable bleibt für alle Zeiten öffentlich. Sie können in vererbten Klassen keine zusätzliche Steuerung hinzufügen. Haben Sie hingegen Ihre Daten nur durch Eigenschaftenprozeduren nach außen offen gelegt, können Sie zu einem späteren Zeitpunkt noch zusätzliche Regeln (Bereichsabfragen, Fehler abfangen) hinzufügen. Sie brauchen dazu die ursprüngliche Klasse kein bisschen zu verändern.

Eigenschaften sind also öffentlichen Feldern in jedem Fall vorzuziehen. Damit das Erstellen von Eigenschaften nicht zu viel Tipparbeit verschlingt, gibt es in Form der Visual Basic-Codeausschnittsbibliothek eine Eingabehilfe, die der folgende Kasten beschreibt.

Zeit sparen beim Erstellen von Eigenschaftenprozeduren mit Code-Ausschnitten

An der Beispielklasse des vorherigen Abschnittes lässt sich eindrucksvoll aufzeigen, dass das Verwalten, Regeln und auch Schützen der Member-Variablen einer Klasse durch entsprechende Eigenschaftenprozeduren durchaus sinnvoll ist, Ihnen jedoch auch eine Menge Tipparbeit abverlangt.

Mithilfe von Codeausschnitten können Sie sich gerade beim »Properties kloppen«, wie es umgangssprachlich in Entwicklerkreisen genannt wird, eine Menge an Zeit sparen. Möchten Sie eine neue Eigenschaft in Ihrer Klasse einführen, die auf einer Member-Variablen basiert, verfahren Sie am besten wie folgt:

- Schreiben Sie die Zeichenfolge **Pro** an die Stelle, an der Sie die neue Eigenschaftenprozedur erstellen wollen. Sie sehen anschließend einen Dialog, etwa wie auch in Abbildung 4.6 zu sehen.
- Drücken Sie zweimal , um den Property-Codeausschnitt einzufügen. Sie sehen anschließend ein Szenario, etwa wie in Abbildung 4.6 zu sehen.

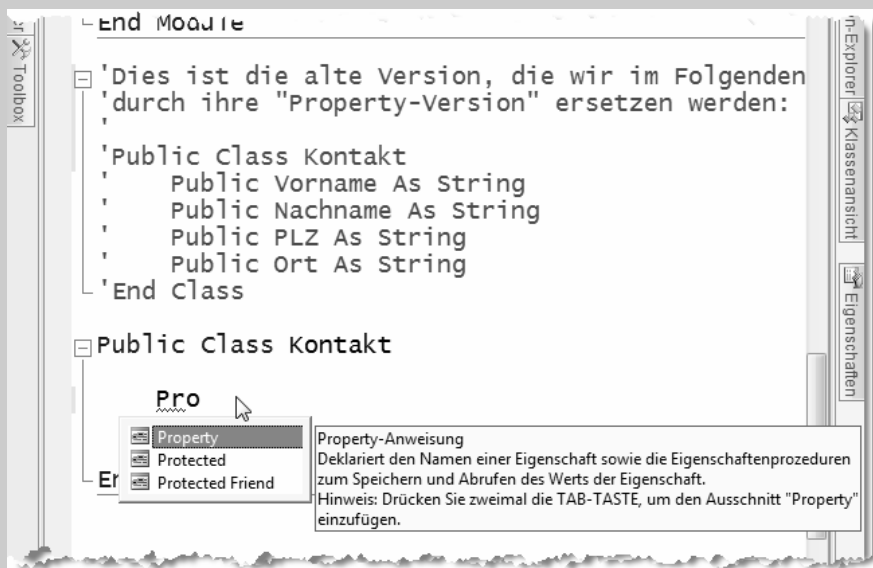






Abbildung 4.6 Um eine Eigenschaftenprozedur durch Codeausschnitte einzufügen, schreiben Sie die Zeichenfolge »Pro«. Drücken Sie, wie im Hinweis der IntelliSense-Liste zu sehen, zweimal , um den Property-Codeausschnitt einzufügen, ...


- Editieren Sie anschließend die Member-Variablen, die zunächst mit dem Vorgabennamen `newPropertyValue` eingefügt wurde, in den Namen, den Sie für die Member-Variablen vorsehen.


```
Public Class Kontakt
    Private newPropertyValue As String
    Public Property NewProperty() As String
        Get
            Return newPropertyValue
        End Get
        Set(ByVal value As String)
            newPropertyValue = value
        End Set
    End Property
End Class
```


Abbildung 4.7 ... um dann anschließend die selektierte Member-Variablen zu editieren und mit  den passenden Datentyp hinzuzufügen ...

- Sobald Sie  gedrückt haben, ändern sich alle Referenzen der Member-Variablen im Codeblock in den neu benannten Namen. Gleichzeitig gelangen Sie zum nächsten editierbaren Feld des Codeausschnitts, dem Typen der Eigenschaftenprozedur. Ändern Sie diesen, und drücken Sie abermals die Taste .

```
Public Class Kontakt
    Private myVorname As String
    Public Property NewProperty() As String
        Get
            Return myVorname
        End Get
        Set(ByVal value As String)
            myVorname = value
        End Set
    End Property
End Class
```

Abbildung 4.8 ... den Sie anschließend im Bedarfsfall ebenfalls editieren, um schließlich mit einem weiteren  das Erstellen der Eigenschaftenprozedur mit dem Editieren des eigentlichen Eigenschaftennamens abzuschließen.

- Sobald Sie  gedrückt haben, ändern sich alle Referenzen der Member-Variablen im Codeblock in den neuen Namen. Gleichzeitig gelangen Sie zum nächsten editierbaren Feld des Codeausschnitts, dem Typen der Eigenschaftenprozedur. ▶

- Ändern Sie diesen, und drücken Sie abermals die Taste . Damit ändern Sie die Typen sowohl für die Eigenschaftenprozedur (und deren value-Variablen mit dem im Set-Accessor der Zuweisungswert an die Eigenschaft übergeben wird) als auch für die korrelierende Member-Variable, die in der Eigenschaftenprozedur gesetzt wird.
- Schließlich ändern Sie noch den Namen der Eigenschaft. Damit sind mit der Definition der Eigenschaftenroutine fertig.

Im Ergebnis stehen alle Eigenschaften und deren entsprechende Member-Variablen jeweils als Pärchen zusammen. Viele Entwickler mögen das nicht in dieser Form; sie bevorzugen, dass Member-Variablen am Anfang der Klassencoddatei definiert werden – Sie können die Member-Variablen natürlich anordnen, wie Sie möchten; das tut der Funktionalität der Klasse keinen Abbruch.

Klassenmethoden mit Sub und Function

Klassencode wird nicht nur in Eigenschaftenprozeduren ausgeführt. Auch Methoden, die in Visual Basic durch Sub oder Function definiert werden, gehören dazu. Methoden nicht statischer Natur (solche also, die nicht, wie in »Statische Methoden« ab Seite 71 beschrieben) sind dabei Methoden, die mit den Member-Variablen einer Klasse arbeiten sollten.

Ein Beispiel dafür ist das folgende: Eine Funktion wie beispielsweise KontaktText könnte für das Zurückliefern der textrepräsentativen Ausgabe der Klasse sorgen – und die entsprechenden Routinen dafür würden dann folgendermaßen aussehen:

```
Public Function KontaktText() As String
    Return Me.Nachname & ", " & Me.Vorname & ", " & _
        Me.PLZ & ", " & Me.Ort
End Function
```

Diese Member-Methode der Klasse bedient sich direkt der Instanzvariablen, und gibt sie verkettet als einen zusammenhängenden String aus. Das bedeutet, dass wir entsprechende Änderungen auch bei der Ausgaberoutine im Modul machen können, die die einzelnen Instanzen verwendet:

```
Sub AdressenAusgeben(ByVal von As Integer, ByVal bis As Integer)
    For c As Integer = von To bis
        Console.WriteLine(c.ToString("000") & ": " & Kontakte(c).KontaktText())
    Next
End Sub
```

Die Methode KontaktText wird hier für jede auszugebenden Kontakt-Instanz aufgerufen und liefert natürlich auch für jede Instanz ein unterschiedliches Ergebnis – hieran sieht man deutlich, dass es sich um eine Member-Methode handelt.

Zugriffsmodifizierer für Klassen, Methoden und Eigenschaften

Zugriffsmodifizierer sind Schlüsselworte, mit denen Sie in Visual Basic bestimmen können, von wo aus der Zugriff auf ein Element gestattet ist und von wo aus nicht. Die Zugriffsmodifizierer `Private` und `Public` haben Sie schon kennen gelernt. Sie bestimmen, ob auf ein Element nur innerhalb eines bestimmten Gültigkeitsbereiches zugegriffen werden kann (`Private`) oder von überall aus (`Public`). Welche weiteren Zugriffsmodifizierer es für Objekte, Klassen und Funktionen/Eigenschaften gibt, zeigen die folgenden Tabellen:

Zugriffsmodifizierer bei Klassen

HINWEIS Wenn nichts anderes gesagt wird, werden Klassen standardmäßig als `Friend` deklariert.

Zugriffsmodifizierer	CTS-Bezeichnung	Beschreibung
Private	Private	Als <code>Private</code> können Klassen nur dann definiert werden, wenn sie geschachtelt in einer anderen Klasse definiert sind. Beispiel: <pre>Public Class A Private Class B End Class End Class Public Class C 'Zugriff verweigert, Class B ist Private! Dim b as A.B End Class</pre>
Public	Public	Sie können auf die Klasse uneingeschränkt von außen zugreifen, auch aus anderen Assemblies heraus.
Friend	Assembly	Sie können innerhalb der Assembly auf die Klasse zugreifen, aber nicht aus einer anderen Assembly heraus.
Protected	Family	Es gilt das für <code>Private</code> Gesagte. Zusätzlich gilt: Auch aus der Klasse abgeleitete Klassen können auf die mit <code>Protected</code> gekennzeichneten und geschachtelten »inneren« Klassen zugreifen.
Protected Friend	FamilyOrAssembly	Der Zugriff auf die geschachtelte Klasse ist in abgeleiteten und von Klassen der gleichen Assembly aus möglich.

Tabelle 4.1 Mögliche Zugriffsmodifizierer für Klassen in Visual Basic .NET

Zugriffsmodifizierer bei Prozeduren (Subs, Functions, Properties)

HINWEIS Wenn nichts anderes gesagt wird, werden `Subs`, `Functions` und `Properties` standardmäßig als `Public` deklariert. Sie sollten gerade bei diesen Elementen aber auf jeden Fall einen Zugriffsmodifizierer verwenden, damit beim Blättern durch den Quellcode schnell deutlich wird, welchen Zugriffsmodus ein Element innehat.

Zugriffsmodifizierer	CTS-Bezeichnung	Beschreibung
Private	Private	Nur innerhalb einer Klasse kann auf die Prozedur zugegriffen werden.
Public	Public	Sie können auf die Prozedur uneingeschränkt von außen zugreifen, auch aus anderen Assemblies heraus.
Friend	Assembly	Sie können innerhalb der Assembly auf die Prozedur zugreifen, aber nicht aus einer anderen Assembly heraus.
Protected	Family	Nur innerhalb der Klasse oder einer abgeleiteten Klasse kann auf die Prozedur zugegriffen werden.
Protected Friend	FamilyOrAssembly	Nur innerhalb der Klasse, einer abgeleiteten Klasse oder innerhalb der Assembly kann auf die Prozedur zugegriffen werden.

Tabelle 4.2 Mögliche Zugriffsmodifizierer für Prozeduren in Visual Basic .NET

Zugriffsmodifizierer bei Variablen

Variablen, die auf Klassenebene nur mit `Dim` deklariert werden, gelten als `Private`, also nur von der Klasse aus zugreifbar. Variablen, die innerhalb eines Codeblocks oder auf Prozedurebene deklariert werden, gelten nur für den entsprechenden Codeblock. Innerhalb eines Codeblocks können Sie nur die `Dim`-Anweisung und keine anderen Zugriffsmodifizierer verwenden. Auf Prozedurebene können Sie eine Variable zusätzlich als `Static` deklarieren. Mehr über den `Static`-Modifizierer erfahren Sie im Abschnitt »Statische Komponenten«.

Zugriffsmodifizierer	CTS-Bezeichnung	Beschreibung
Private	Private	Nur innerhalb einer Klasse kann auf die Variable zugegriffen werden. Variablen innerhalb von Prozeduren oder noch kleineren Gültigkeitsbereichen können nicht explizit als <code>Private</code> definiert werden, sind es aber standardmäßig.
Public	Public	Von außen kann auf die Klassenvariable uneingeschränkt zugegriffen werden. Sie sollten Variablen aber bestenfalls als <code>Protected</code> deklarieren und sie nur durch Eigenschaften nach außen offen legen. Mehr zu diesem Thema erfahren Sie im Abschnitt »Öffentliche Variablen oder Eigenschaften – eine Glaubensfrage?« auf Seite 63.
Friend	Assembly	Sie können innerhalb der Assembly auf die Klassenvariable zugreifen, aber nicht aus einer anderen Assembly heraus. Es gilt das für <code>Public</code> Gesagte.
Protected	Family	Nur innerhalb derselben oder einer abgeleiteten Klasse kann auf die Variable zugegriffen werden. Variablen sollten in Klassen, bei denen Sie davon ausgehen, dass sie später des Öfteren vererbt werden, als <code>Protected</code> definiert werden, damit abgeleitete Klassen ebenfalls darauf zugreifen können.
Protected Friend	FamilyOrAssembly	Nur innerhalb der Klasse, einer abgeleiteten Klasse oder innerhalb der Assembly kann auf die Klassenvariable zugegriffen werden. Von dieser Kombination sollten Sie absehen.
Static	- - -	Sonderfall in Visual Basic. Lesen Sie dazu bitte die Ausführungen im Abschnitt »Statische Komponenten«.

Tabelle 4.3 Mögliche Zugriffsmodifizierer für Variablen in Visual Basic .NET

Diese Tabellen sollen Ihnen kompakt und auf einen Blick die Zugriffsmodifizierer verdeutlichen. Die CTS-Bezeichnungen der Zugriffsmodifizierer benötigen Sie, wenn Sie sich den IML-Code einer Klasse anschauen, um zu erkennen, welchen Zugriffsmodus beispielsweise eine Methode hat.

Unterschiedliche Zugriffsmodifizierer für Eigenschaften-Accessors

Seit Visual Studio 2005 ist es möglich, dass die Get- und Set-Accessors unterschiedliche Zugriffsmodifizierer aufweisen. So haben Sie beispielsweise die Möglichkeit, zu bestimmen, dass zwar eine bestimmte Eigenschaft von jedem Ort aus gelesen werden kann (Public), aber Eigenschaften nur von derselben Klasse aus geschrieben werden dürfen (Private). Im Code würde eine solche Eigenschaft folgendermaßen ausschauen:

```
Module Module1
    Sub Main()
        Dim locEigenschaftenTest As New EigenschaftenTest("Text für Eigenschaft")

        'Das Auslesen der Eigenschaft ist problemlos möglich
        Console.WriteLine("Eigenschaft enthält: " & locEigenschaftenTest.EineEigenschaft)

        'FEHLER: Der Set-Zugriffsmodifizierer verbietet aber das Schreiben,
        'weil er 'private' ist.
        locEigenschaftenTest.EineEigenschaft = "Neuer Text"
    End Sub
End Module

Public Class EigenschaftenTest

    Private myEineEigenschaft As String

    Sub New(ByVal textFürEigenschaft As String)
        'Ist erlaubt - die Klasse darf die
        'Eigenschaft beschreiben!
        EineEigenschaft = textFürEigenschaft
    End Sub

    Public Property EineEigenschaft() As String
        Get
            Return myEineEigenschaft
        End Get
        Private Set(ByVal value As String)
            myEineEigenschaft = value
        End Set
    End Property
End Class
```

Dieses kleine Beispiel besteht aus zwei Einheiten – einem Modul und einer Klasse. Die Klasse EigenschaftenTest enthält einen parametrisierten Konstruktor sowie eine Eigenschaft, die aber Accessoren mit unterschiedlichen Zugriffsmodifizierern hat.

Innerhalb des Konstruktors (`Sub New`) ist der Schreibzugriff auf die Eigenschaft problemlos möglich, da aus der Klasse selbst heraus auch auf Elemente zugegriffen werden kann, deren Zugriff mit `Private` eingeschränkt wurde. Innerhalb des Moduls funktioniert der Zugriff allerdings nicht mehr, da sich das Programm zum Zeitpunkt des Zugriffs außerhalb der Klasse befindet, und wegen `Private` ist von diesem Punkt aus kein Herankommen an die Eigenschaft möglich.

Doch wozu brauchen Sie unterschiedliche Zugriffsmodifizierer in Eigenschaften während der Entwicklung Ihrer Software? Denken Sie beispielsweise an das zur Verfügung stellen von Assemblies (Klassenbibliotheken) für anderer Entwickler: Sie möchten beispielsweise in der Lage sein, bestimmte Eigenschaften Ihrer Klassen von jedem Punkt Ihrer Assembly aus zu manipulieren; Sie möchten aber gleichzeitig verhindern, dass ein Entwickler, der Ihre Assembly verwendet, dazu auch in der Lage ist. In diesem Fall definieren Sie den `Get`-Accessor als `Public` – das Lesen der Eigenschaft stellt schließlich kein Risiko dar und kann von überall aus erfolgen – aber den `Set`-Accessor als `Friend`. Vom gesamten Programmcode, der sich in Ihrer Klassenbibliothek befindet, können Sie dann die Eigenschaft der betreffenden Klasse manipulieren; Entwickler, die die Assembly einbinden, also von außerhalb Ihrer Assembly zugreifen, können Sie aber nicht mehr direkt manipulieren. Solcherlei Eigenschaften sind dann wichtig, wenn es sich bei ihnen um Quasi-Konstanten handeln soll. Ihre Assembly definiert die Eigenschaft wann und wie sie will auf Grund bestimmter Zustände. Die Assemblies, die sie konsumieren, müssen aber mit dieser Einstellung leben. Klassen, die beispielsweise Einstellungen aus der Windows-Registry widerspiegeln, können davon Gebrauch machen. Denkbar wäre auch, eine Verbindungszeichenfolge zu einem SQL Server auf diese Weise freizulegen – Sie hätten zwar aus Ihrer Assembly heraus Manipulationsspielraum für die Verbindungszeichenfolge; eine Assembly könnte aber immer nur die Verbindungszeichenfolge mit der Eigenschaft auslesen, die Sie innerhalb Ihrer Assembly vorgeben.

Vererbung, Polymorphie, Abstrakte Klassen und Schnittstellen

Um es ganz kurz zu machen: Die Schlagworte, die Sie in dieser Überschrift finden, würden den Rahmen dieses Buches sprengen. Sie finden in diesem Kapitel, wie bereits angekündigt, ausschließlich neue Features und solche Dinge erklärt, die Sie als Grundlagen für weitere neue Features, insbesondere für *LINQ to Objects* benötigen.

Falls Sie an weiterem, ausführlichem Lehrmaterial zur objektorientierten Programmierung interessiert sind, empfehle ich Ihnen das Buch *Visual Basic 2005 – Das Entwicklerbuch*, das Sie auf der ActiveDevelop-Internetseite <http://www.activedevelop.de> inklusive aller Beispiele kostenlos herunterladen können.

Kurz zusammengefasst handelt es sich bei den einzelnen Themen um Folgendes:

- **Klassenvererbung** erfolgt durch das `Inherits`-Schlüsselwort. Damit haben Sie die Möglichkeit, vorhandene Klassen – ganz gleich, ob das Ihre eigenen oder Klassen aus bereits vorhandenen Klassenbibliotheken (und damit auch des .NET Frameworks) sind – zu erweitern.
- **Polymorphie** kann durch das Überschreiben von besonders gekennzeichneten Eigenschaften und Methoden realisiert werden. Dadurch haben Sie die Möglichkeit, Verhaltensweisen von Klassen beim Vererben nicht nur zu übernehmen, sondern auch gezielt abzuändern oder zu erweitern.

- **Abstrakte Klassen** sind Klassen, die nur zur Implementierung von Grundfunktionalitäten für das Vererben in andere Klassen vorgesehen sind – etwa wie eine Dokumentenvorlage für weitere Word-Dokumente. Abstrakte Klassen können selbst nicht instanziiert werden.
- **Schnittstellen (Interfaces)** schreiben vor, welche Elemente (Methoden, Eigenschaften, Ereignisse, etc.) eine Klasse zu implementieren hat. Sie dienen ebenfalls dazu, eine Klasse für bestimmte Aufgaben oder Möglichkeiten zu kennzeichnen.

Statische Komponenten

Bislang haben Sie im Rahmen von Klassen nur Member-Variablen (Felder), Member-Eigenschaften und Member-Methoden kennengelernt, also solche Methoden, die zum sauberen Funktionieren auf die Instanzvariablen zugreifen mussten, wofür sie in jedem Fall eine Instanz der Klasse benötigten – anderenfalls hätten Sie eine `NullReferenceException` ausgelöst, wie im Abschnitt »Nothing« ab Seite 57 zu lesen.

Sie haben aber auch die Möglichkeit, statische Komponenten zu erstellen, die eben nicht auf Member-Variablen einer Klasse zurückgreifen, und damit auch ohne Instanziierung aufrufbar sind. Die folgenden Abschnitte zeigen, wie's geht:

Nicht durch die Schlüsselwörter `Static` und `Shared` in Visual Basic verwirren lassen!

`Static` im Gegensatz zu `Shared` bewirkt, dass eine Variable, die nur innerhalb einer Prozedur (Sub oder Function) verwendet wird, ihren Inhalt auch nach Verlassen der Unterroutine nicht verliert. Dennoch können Sie auf diese Variable nur innerhalb der Unterroutine zugreifen, in der sie definiert wird. Im Prinzip ist eine mit `Static` definierte Variable eine, die als `Shared`-Member für die Klasse definiert und mit einem internen Attribut versehen wurde, das die Verwendung auf den Gültigkeitsbereich reglementiert, in dem sie deklariert wurde.

Vorhanden sind `Static`-Variablen übrigens nur aus Gründen der Kompatibilität zum alten Visual Basic 6.0 (und vorherigen Versionen).

Statische Methoden

Statische Methoden sind Methoden (Sub, Function) die mit dem Schlüsselwort `Shared` (nicht `Static`! – Siehe vorheriger Kasten!) gekennzeichnet und damit direkt über die Klasse und nicht über die Klasseninstanz definiert sind. Das bedeutet: Anders als bei Instanzmethoden können Sie eine statische Methode direkt über den Klassennamen aufrufen und brauchen eben keine Instanz dafür.

Ein gutes Beispiel für eine statische Methode ist die `Parse`-Methode aller numerischen Datentypen. Möchten Sie beispielsweise eine Zeichenfolge in einen `Integer`-Wert konvertieren, bedienen Sie sich genau dieser Funktion:

```
Dim intVar as Integer = Integer.Parse("1234")
```

Hier sehen Sie, dass Sie sich für den Funktionsaufruf keiner Instanzmethode bedienen, da auch keine Member-Variablen der Instanz für den Konvertierungsprozess benötigt werden. Im umgekehrten Fall ist das anders. Möchten Sie einen `Integer`-Wert in eine Zeichenkette umwandeln, bedienen Sie sich der Instanzmethode `ToString`, die natürlich auf die interne Member-Variable des `Integer`-Datentyps zurückgreift:

```
Dim intVar as Integer = 5
Debug.Print(intVar.ToString())
```

HINWEIS Im Grunde genommen müsste die `EllipseString`-Methode unseres Klassenbeispiels ebenfalls statisch und damit mit `Shared` definiert werden, da sie ebenfalls nur einen Algorithmus kapselt, der auf *keine* Member-Variablen der Klasse direkt zugreift. Konsequenterweise sollte diese Methode dann auch als öffentlich definiert werden, sodass ihre korrekte Signatur folgendermaßen aussehen sollte:

```
Public Shared Function EllipseString(ByVal text As String, ByVal MaxLength As Integer) As String
    .
    .
    .
End Function
```

Die richtige Funktionsweise dieser Routine wäre dann:

```
Dim s as String = Kontakt.EllipseString("Dies ist die abzuschneidende Zeichenkette", 20)
```

Sollten Sie eine Objektvariable auf eine instanziierte `Kontakt`-Instanz haben, könnten Sie rein theoretisch auch über diese Objektvariable auf die statische Methode zugreifen. Doch Sie sollten das nicht machen, da es für Verwirrung sorgt. Die Verwendung von Objektvariablen impliziert immer, dass Sie etwas mit der Klasseninstanz (also dem Dateninhalt der Klasse) anstellen wollen, was Sie in diesem Fall gar nicht machen, da, wie gerade gesagt, statische Methoden keine Member-Variablen nutzen (und auch gar nicht nutzen können!). Dieser Meinung ist auch der Background-Compiler von Visual Basic, denn einen Versuch, eine statische Methode über eine Objektvariable aufzurufen, quittiert er, wie in Abbildung 4.9 zu sehen, mit einer Warnung.

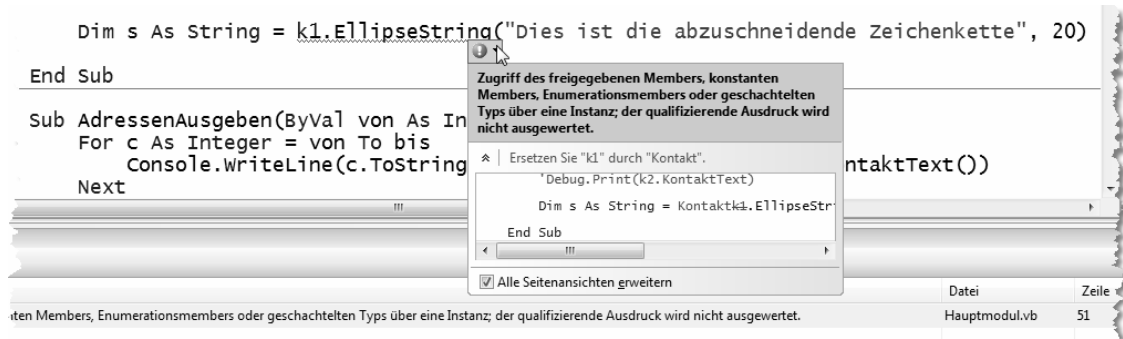


Abbildung 4.9 Einen Versuch, eine statische Methode über eine Objektvariable aufzurufen, quittiert der Background-Compiler von Visual Basic mit einer Warnung. Die Fehlerkorrekturoptionen helfen Ihnen übrigens beim Berichtigen.

Statische Eigenschaften

Visual Basic sieht auch statische Eigenschaften vor. Genau wie bei statischen Funktionen sind statische Eigenschaften direkt über die Klasse und nicht über die Klasseninstanz definiert. Das bedeutet, dass statische Eigenschaften Funktionalitäten bereitstellen sollten, die für alle Objekte dieser Klasse gelten und nicht für eine bestimmte Objektinstanz.

Das beste Beispiel für eine statische Eigenschaft ist die `Now`-Eigenschaft von `DateTime`. Sie liefert die aktuelle Uhrzeit zurück und ist natürlich von den Eigenschaften einzelner `DateTime`-Instanzen völlig unabhängig. Statische Eigenschaften werden, ebenfalls wie statische Funktionen, mit dem `Shared`-Schlüsselwort deklariert. Ein Beispiel für eine statische Eigenschaft finden Sie im folgenden Abschnitt.

Module in Visual Basic – automatisch statische Klassen erstellen

Module gibt es bei allen bislang existierenden .NET-Programmiersprachen nur in Visual Basic. Und auch hier ist ein Modul nichts weiter als eine Mogelpackung, denn das, was als Modul bezeichnet wird, ist im Grunde genommen nichts anderes als eine abstrakte Klasse (eine, die also nicht instanziiert ist) mit ausschließlich statischen Methoden, statischen Eigenschaften und statischen öffentlichen Feldern.

Halten wir fest:

- Ein Modul ist nicht instanziiert. Eine abstrakte Klasse auch nicht.
- Ein Modul kann keine überschreibbaren Prozeduren zur Verfügung stellen. Die statischen Prozeduren einer Klasse können das auch nicht.
- Ein Modul kann nur Prozeduren zur Verfügung stellen, auf die aber nur ohne Instanzobjekt direkt zugegriffen werden kann. Das gleiche gilt für die statischen Prozeduren einer abstrakten Basisklasse.

Es gibt aber auch feine Unterschiede: So können Module beispielsweise keine Schnittstellen implementieren; das können zwar abstrakte Klassen, aber Sie können keine statischen Schnittstellenelemente definieren. Insofern ist dieser scheinbare Unterschied in Wirklichkeit gar keiner. Ein Modul kann auch nur auf oberster Ebene definiert und nicht in einem anderen geschachtelt werden.

Module setzen Sie bei der OOP vorschlagsweise so wenig wie möglich ein, denn sie widersprechen dem Anspruch von .NET, möglichst wieder verwendbaren Code zu schaffen.

Hier im Buch finden Sie aus diesem Grund Module nur, wenn es um »Quick-And-Dirty«-Projekte geht, bei denen beispielsweise eine Konsolen-Anwendung Tests durchzuführen hat oder »mal eben« etwas demonstrieren soll, genauso, wie Sie es in den vergangenen Kapiteln bereits erlebt haben.

Delegaten und Lambda-Ausdrücke

Lambda-Ausdrücke sind quasi anonyme Funktionen, die Ausdrücke und Anweisungen enthalten und für die Erstellung von Delegaten oder Ausdrucksbaumstrukturen verwendet werden können.

Gerade wieder mit Schwerpunkt auf LINQ ist es oft notwendig, an bestimmten Stellen Delegaten – also Funktionszeiger – einzusetzen, die aber ausschließlich bei einem konkreten Aufruf und mit einer Minimalausstattung an Code zurecht kommen.

Erinnern wir uns: Delegaten sind Typen, die Adressen speichern – ähnlich wie Objektvariablen Zeiger auf Instanzen von Klassen. Allerdings speichern Delegaten keine Speicheradressen auf Objektdaten im Managed Heap, sondern die Speicheradressen von Methoden, die dann, wie eine Sub oder Function selbst, über die Delegaten aufgerufen werden können. Das praktische daran: Eine Delegatenvariable kann mal auf die eine oder mal auf die andere Methode zeigen – je nachdem, welche Methode eben im entsprechenden Kontext Sinn machen. Wichtig nur: Die Methoden müssen dabei die gleiche *Signatur* besitzen, also die gleichen Typen von Parametern in der gleichen Reihenfolge übernehmen.

Umgang mit Delegation am Beispiel

Schauen wir uns eine weitere Version unseres bisherigen Beispiels an, zunächst um ein Gefühl für die Verwendung von Delegation zu bekommen. Wir bauen dessen Sortierroutine dort so um, dass der Sortierkriteriumsausdruck, der bislang fest im Programm mit dem Nachnamen des Kontakts verdrahtet war, durch einen Delegaten ersetzt wird. Der Delegat wiederum erwartet zwei Parameter, nämlich die Kontakte, die miteinander verglichen werden sollen. Den Rückgabewert liefert er vom Typ Integer zurück, mit folgendem Hintergrund. Ergibt das Funktionsergebnis 0, sind beide Kontakte gleich, bei 1, ist der erste größer, bei -1 ist der erste kleiner. Was letzten Endes verglichen wird, regelt dann eine Prozedur, auf die die Delegatenvariable zeigen soll. Die Änderungen dazu sehen zunächst einmal folgendermaßen aus:

```
Public Delegate Function ComparerDelegate(ByVal k1 As Kontakt, ByVal k2 As Kontakt) As Integer

Sub AdressenSortieren(ByVal cDel As ComparerDelegate)

    Dim anzahlElemente As Integer = 101

    Dim aeussererZaehler As Integer
    Dim innererZaehler As Integer
    Dim delta As Integer

    Dim tempKontakt As Kontakt

    delta = 1

    'Größten Wert der Distanzfolge ermitteln
    Do
        delta = 3 * delta + 1
    Loop Until delta > anzahlElemente

    Do
        'Ist Abbruchkriterium - deswegen herunterzählen
        delta \= 3

        'Shellsort's Kernalgorithmus
        For aeussererZaehler = delta To anzahlElemente - 1
            tempKontakt = Kontakte(aeussererZaehler)

            innererZaehler = aeussererZaehler
            Do
                If cDel.Invoke(tempKontakt, Kontakte(innererZaehler - delta)) = 1 OrElse _
                    cDel.Invoke(tempKontakt, Kontakte(innererZaehler - delta)) = 0 Then Exit Do
                Kontakte(innererZaehler) = Kontakte(innererZaehler - delta)

                innererZaehler = innererZaehler - delta
                If (innererZaehler <= delta) Then Exit Do
            Loop
            Kontakte(innererZaehler) = tempKontakt
        Next
    Loop Until delta = 0
End Sub
```

An den zuletzt in Fettschrift formatierten Zeilen dieses Listings können Sie erkennen, wie der eigentliche Aufruf einer Methode erfolgt, die durch eine Delegatenvariable referenziert wird – nämlich mit der Invoke-Methode, der genau die Parameter übergeben werden, die sie normalerweise der Methode übergeben würde.

Und jetzt schauen wir uns an, wie wir uns diesen Umbau für eine flexiblere Nutzung der Sortieroutine zu Nutze machen können:

```
Sub Main()  
.  
.  
.  
    'Zufallsadressen generieren  
    Console.WriteLine("Zufallsadressen werden generiert ... ")  
    ZufallsAdressenGenerieren()  
    Console.WriteLine("fertig!")  
  
    'Die ersten 10 Zufallsadressen ausgeben  
    AdressenAusgeben(0, 10)  
  
    'Die Adressen nach Nachnamen sortieren  
    Console.WriteLine()  
    Console.WriteLine("Adressen werden nach Nachnamen sortiert ... ")  
    Dim compDelegate As ComparerDelegate = AddressOf KontaktNachnamenVergleich  
    AdressenSortieren(compDelegate)  
    Console.WriteLine("fertig!")  
    Console.WriteLine()  
  
    'Die ersten 10 Zufallsadressen ausgeben  
    AdressenAusgeben(0, 10)  
  
    'Die Adressen nach Ortsnamen sortieren  
    Console.WriteLine()  
    Console.WriteLine("Adressen werden nach Ortsnamen sortiert ... ")  
    compDelegate = AddressOf KontaktOrtVergleich  
    AdressenSortieren(compDelegate)  
    Console.WriteLine("fertig!")  
    Console.WriteLine()  
.  
.  
.  
End Sub
```

Hier werden jetzt an zwei Stellen der Sortieroutine die Vergleichsdelegaten übergeben, mit denen dann die eigentliche Sortierung durchgeführt wird. Diese beiden Methoden schauen folgendermaßen aus:

```
Function KontaktNachnamenVergleich(ByVal k1 As Kontakt, ByVal k2 As Kontakt) As Integer  
    Return String.Compare(k1.Nachname, k2.Nachname)  
End Function  
  
Function KontaktOrtVergleich(ByVal k1 As Kontakt, ByVal k2 As Kontakt) As Integer  
    Return String.Compare(k1.Ort, k2.Ort)  
End Function
```

Soweit, so gut. Und was passiert jetzt genau, wenn das Programm gestartet wird? Die folgende Schritt-für-Schritt-Erklärung macht es deutlich:

1. Nachdem die Zufallsadressen erstellt wurden, definiert die Anwendung die Delegatenvariable `compDelegate` und weist dieser gleichzeitig die Adresse der `KontaktNachnamenVergleich`-Methode zu.
2. Sie ruft die Methode `AdressenSortieren` auf und übergibt ihr gleichzeitig die Delegatenvariable.
3. `AdressenSortieren` wiederum führt `Invoke` auf die Delegatenvariable aus und ruft dabei für den Elementvergleich die Methode `KontaktNachnamenVergleich` auf. Die Kontaktliste wird damit nach Nachnamen sortiert.
4. Wieder zurück im Hauptmodul wird der Delegatenvariable `compDelegate` nach der Sortierung die Adresse der `KontaktOrtVergleich`-Methode zugewiesen.
5. Die Anwendung ruft abermals die Methode `AdressenSortieren` auf und übergibt ihr gleichzeitig die Delegatenvariable, die jetzt allerdings auf eine andere Methode als zuvor zeigt.
6. `AdressenSortieren` führt `Invoke` auf die Delegatenvariable aus und ruft aber dieses Mal dabei für den Elementvergleich die Methode `KontaktOrtVergleich` auf. Die Kontaktliste wird damit beim zweiten Durchlauf nach dem Ortsnamen sortiert.

Lambda-Ausdrücke

Bei einem Lambda-Ausdruck handelt es sich um eine quasi namenlose Funktion, die ein einzelnes Ergebnis zurückliefert. Lambda-Ausdrücke können an allen Stellen verwendet werden, an denen ein Delegationstyp gültig ist.

Das folgende Beispiel stellt einen Lambda-Ausdruck dar, der das ihm übergebene Argument um eins erhöht und den Wert als Funktionsergebnis zurückliefert.

```
Function (num As Integer) num + 1
```

So für sich steht dieser Lambda-Ausdruck recht »lose im Raum«. Zu seiner wiederholten Verwendung kann in vereinfachter Form durch lokalen Typenrückschluss (siehe Kapitel 3) eine Delegatenvariable definiert werden, die die Adresse dieses Lambda-Ausdrucks aufnimmt:

```
Dim add1 = Function(num As Integer) num + 1
```

Im Weiteren können Sie dann für den Aufruf diese Variable verwenden, wobei Sie den Wert in gewohnter Weise als Parameter übergeben.

```
'Die folgende Codezeile bewirkt, dass 6 ausgegeben wird:  
Console.WriteLine(add1(5))
```

Alternativ kann die Funktion gleichzeitig deklariert und ausgeführt werden.

```
Console.WriteLine((Function(num As Integer) num + 1)(5))
```

Ein Lambda-Ausdruck kann als Wert eines Funktionsaufrufs zurückgegeben oder als Argument an einen Delegatenparameter übergeben werden. Im folgenden Beispiel werden boolesche Lambda-Ausdrücke als Argument an die `testResult`-Methode übergeben. Mit dieser Methode wird der boolesche Test auf ein Ganzzahlenargument, `value`, angewendet. Wenn der Lambda-Ausdruck `True` zurückgibt, wird »Erfolgreich« angezeigt, falls er `False` zurückgibt, wird »Fehlgeschlagen« angezeigt.

```
Module Module2

    Sub Main()
        ' Die folgende Zeile gibt 'Erfolgreich' aus, da 4 gerade ist.
        testResult(4, Function(num) num Mod 2 = 0)
        ' Die folgende Zeile gibt 'Fehlgeschlagen' aus, da 5 nicht > 10 entspricht.
        testResult(5, Function(num) num > 10)
    End Sub

    Sub testResult(ByVal value As Integer, ByVal fun As Func(Of Integer, Boolean))
        If fun(value) Then
            Console.WriteLine("Erfolgreich")
        Else
            Console.WriteLine("Fehl geschlagen")
        End If
    End Sub

End Module
```

Lambda-Ausdrücke am Beispiel

Unser uns permanent begleitendes Beispiel können wir mithilfe von Lambda-Ausdrücken stark vereinfachen, da wir den Zwischenschritt der Delegatenvariablendefinition nicht mehr benötigen, sondern der Sortierroutine direkt den Sortierkriteriumsausdruck als Lambda-Ausdruck übergeben können.

```
Sub Main()
    .
    .
    .
    'Die Adressen nach Nachnamen sortieren
    Console.WriteLine()
    Console.WriteLine("Adressen werden nach Nachnamen sortiert ... ")
    AdressenSortieren((Function(k1 As Kontakt, k2 As Kontakt) _
        String.Compare(k1.Nachname, k2.Nachname)))
    Console.WriteLine("fertig!")
    Console.WriteLine()

    'Die ersten 10 Zufallsadressen ausgeben
    AdressenAusgeben(0, 10)

    'Die Adressen nach Ortsnamen sortieren
    Console.WriteLine()
    Console.WriteLine("Adressen werden nach Ortsnamen sortiert ... ")
    AdressenSortieren((Function(k1 As Kontakt, k2 As Kontakt) _
        String.Compare(k1.Ort, k2.Ort)))
    Console.WriteLine("fertig!")
    Console.WriteLine()
    .
    .
    .
End Sub
```

Die Sortieroutine kann diesen Lambda-Ausdruck ebenfalls direkt als solchen entgegennehmen und braucht ebenfalls nicht mehr den »Umweg« über einen definierten Delegationstyp zu gehen:

```
'Lambda-Funktion nimmt zwei Kontakt-Objekte entgegen und liefert Integer zurück
Sub AdressenSortieren(ByVal cDel As Func(Of Kontakt, Kontakt, Integer))

    Dim anzahlElemente As Integer = 101

    Dim aeussererZaehler As Integer
    Dim innererZaehler As Integer
    Dim delta As Integer

    Dim tempKontakt As Kontakt

    delta = 1

    'Größten Wert der Distanzfolge ermitteln
    Do
        delta = 3 * delta + 1
    Loop Until delta > anzahlElemente

    Do
        'Ist Abbruchkriterium - deswegen herunterzählen
        delta \= 3

        'Shellsort's Kernalgorithmus
        For aeussererZaehler = delta To anzahlElemente - 1
            tempKontakt = Kontakte(aeussererZaehler)

            innererZaehler = aeussererZaehler
            Do
                'Invoke ist optional! - Es geht auch ohne:
                If cDel(tempKontakt, Kontakte(innererZaehler - delta)) = 1 OrElse _
                    cDel(tempKontakt, Kontakte(innererZaehler - delta)) = 0 Then Exit Do
                Kontakte(innererZaehler) = Kontakte(innererZaehler - delta)

                innererZaehler = innererZaehler - delta
                If (innererZaehler <= delta) Then Exit Do
            Loop
            Kontakte(innererZaehler) = tempKontakt
        Next
    Loop Until delta = 0
End Sub
```

HINWEIS

Der Einsatz von Lambda-Ausdrücken macht insbesondere dort Sinn, wo Sie es mit so genannten Predicate-, Comparison- oder Action-Delegaten zu tun haben, und das ist sowohl bei der nichtgenerischen `ArrayList` als auch bei der generischen `List(Of)`-Auflistung der Fall. Kapitel 6 hält mehr darüber im letzten Abschnitt bereit. Dort finden Sie auch im Rahmen eines etwas ausführlicheren `List(Of)`-Beispiels entsprechenden Beispielcode dazu.